VCVio: Verified Cryptography in Lean via Oracle Effects and Handlers

Quang Dao
CMU

Work done with Devon Tuma, Alex Hicks, and open-source contributors

Software Verification in Lean, April 21 2026

Formally Verifying Cryptography is Hard

Cryptography protocols are complex

(stateful, probabilistic, interactive, adversarial)

Proofs about crypto protocols range from

rigorous (often difficult), to informal, to non-existent

End-to-end proofs require bridging two worlds

abstract math $\Longrightarrow$ low-level ISA semantics

Formal models always play catch up with reality

(bad case: verification theatre / facade)

Existing Approaches & Tradeoffs

Tool (non-exhaustive)	Model	Foundational	Program Logic	Open Source	Automation & Tactics	Other Notes
Tamarin / ProVerif / etc.	symbolic	no	MSR / Horn	yes	auto + interactive	unbounded sessions, attack finding, equational theories
CryptoVerif	comp.	no	game transforms	yes	auto + interactive	no attack search, narrow scope (protocol proving only)
Easycrypt	comp.	no	eRHL	yes	SMT + relational	SMT integration, Jasmin link, mature
FCF (Rocq)	comp.	yes	n/a	yes	n/a	precursor to VCVio, more cumbersome monad stack
CryptHOL (Isabelle)	comp.	yes	n/a	yes	auto + sledgehammer	coinductive GPV, several AFP entries (constructive crypto, MPC, kyber, etc.)
SSProve (Rocq)	comp.	yes	pRHL	yes	relational + SSP	Hacspec integration (“last yard”), SSP framework
CatCrypt (Lean)	comp.	yes	unclear	no	unclear	(claimed) 172 protocols & constructions formalized, 110 have Rust-to-Lean pipeline
VCVio (Lean)	**comp.**	**yes**	**eRHL**	**yes**	**mvcgen + extensions**	**Base for ArkLib (verifying SNARKs) Pilot of SSP, Hax integration, UC, etc.**

Existing Approaches & Tradeoffs

Tool (non-exhaustive)	Model	Foundational	Program Logic	Open Source	Automation & Tactics	Other Notes
Tamarin / ProVerif / etc.	symbolic	no	MSR / Horn	yes	auto + interactive	unbounded sessions, attack finding, equational theories
CryptoVerif	comp.	no	game transforms	yes	auto + interactive	no attack search, narrow scope (protocol proving only)
Easycrypt	comp.	no				SMT integration, Jasmin link, mature
FCF (Rocq)	comp.	yes				precursor to VCVio, more cumbersome monad stack
CryptHOL (Isabelle)	comp.	yes	n/a	yes	auto + sledgehammer	coinductive GPV, several AFP entries (constructive crypto, MPC, kyber, etc.)
SSProve (Rocq)	comp.	yes	pRHL	yes	relational + SSP	Hacspec integration (“last yard”), SSP framework
CatCrypt (Lean)	comp.	yes				(claimed) 172 protocols & constructions formalized, 110 have Rust-to-Lean pipeline
VCVio (Lean)	**comp.**	**yes**				**Base for ArkLib (verifying SNARKs) Pilot of SSP, Hax integration, UC, etc.**

Before AI

After AI

VCVio: Verified Cryptography via Algebraic Effects



VCV-io Public

Observation: cryptographic protocols depend crucially on **oracle calls**

- to sample randomness, perform hashing, interact with other parties, etc.

Design aesthetics:

- Minimal, most general base layer. Extensions are conservative and used when needed
- Grounded in PL theory, category theory, and formal verification literature

$\Longrightarrow$ oracle semantics through the lens of algebraic effects and handlers

Oracle Computations as Free Monads

```
/-- Response types indexed by query value. -/
def OracleSpec (ι : Type u) : Type _ := ι → Type v

/-- The free monad over the polynomial functor `spec`. -/
inductive OracleComp (spec : OracleSpec ι) : Type w → Type _ where
  | pure : α → OracleComp spec α
  | queryBind : (i : ι) → (spec i → OracleComp spec α) → OracleComp spec α

/-- Probabilistic computation with uniform-sampling oracle. -/
abbrev ProbComp (α : Type) : Type := OracleComp (fun n : ℕ => Fin (n + 1)) α
```

Failure, state, exception, etc. layered on top of OracleComp **as needed**

```
/-- Embedding of hax/aeneas: `Rust.Error` for panics, `OptionT` for divergence. -/
abbrev RustOracleComp (spec : OracleSpec ι) (α : Type) : Type :=
  ExceptT Rust.Error (OptionT (OracleComp spec)) α
```

Oracle Simulation as Effect Handlers

```
/-- A handler: how to interpret each oracle query in the monad `m`. -/
def QueryImpl (spec : OracleSpec ι) (m : Type → Type) := (i : ι) → m (spec i)

/-- Run an `OracleComp` by substituting each query with its handler in `m`. -/
def simulateQ [Monad m] (impl : QueryImpl spec m) : OracleComp spec α → m α
  | .pure a        => pure a
  | .queryBind i k => impl i >>= fun u => simulateQ impl (k u)
```

```
/-- Side observation per query: write a trace into `WriterT ω m`.
Specializes to logging and query accounting. -/
def withTrace [Monoid ω] (impl : QueryImpl spec m)
    (traceFn : (t : spec.Domain) → spec.Range t → ω) :
    QueryImpl spec (WriterT ω m) :=
  fun t => do let u ← impl t; tell (traceFn t u); return u

/-- Lazy random oracle: replay cached response, sample fresh on miss. -/
def withCaching (impl : QueryImpl spec m) : QueryImpl spec (StateT (QueryCache spec) m) :=
  fun t => do match (← get) t with
    | some u => return u
    | none   => let u ← impl t; modifyGet fun c => (u, Function.update c t (some u))
```

Denotational Semantics & Program Logic

```
/-- Subprobability denotation: monads with possible failure mass. -/
class HasEvalSPMF (m : Type → Type) [Monad m] where
  toSPMF : m →ᵐ SPMF

/-- Total probability denotation: failure-free monads. -/
class HasEvalPMF (m : Type → Type) [Monad m] where
  toPMF : m →ᵐ PMF

/-- User-facing denotation: project through the typeclass. -/
abbrev evalDist [HasEvalSPMF m] : m α → SPMF α := HasEvalSPMF.toSPMF

/-- `OracleComp` is failure-free: each query is a fresh uniform sample. -/
instance [spec.Sampleable] : HasEvalPMF (OracleComp spec) where
  toPMF := simulateQ (fun t => PMF.uniformOfFintype (spec.Range t))

/-- Failure from `OptionT`: lift any `HasEvalSPMF m` through the transformer. -/
instance [HasEvalSPMF m] : HasEvalSPMF (OptionT m) where
  toSPMF := OptionT.mapM' HasEvalSPMF.toSPMF
```

Typeclasses allow for more general definitions, not necessarily tied to OracleComp

Denotational Semantics & Program Logic

```
/-- Relational weakest precondition: best expected `g` over any coupling of the two outputs. -/
def eRelWP (oa : OracleComp spec₁ α) (ob : OracleComp spec₂ β) (g : α → β → ℝ≥0∞) : ℝ≥0∞ :=
  ⨆ μ : SPMF.Coupling (evalDist oa) (evalDist ob), ∑' z, Pr[= z | μ.1] * g z.1 z.2

/-- eRHL triple. pRHL is the special case `pre = 1`, `post = indicator R`. -/
def eRelTriple (pre : ℝ≥0∞) (oa : OracleComp spec₁ α) (ob : OracleComp spec₂ β)
    (post : α → β → ℝ≥0∞) : Prop :=
  pre ≤ eRelWP oa ob post
```

[**ABDG25**] A Quantitative Probabilistic Relational Hoare Logic (eRHL). POPL 2025

```
/-- Relational WP algebra parametric in truth lattice `l`. -/
class MAlgRelOrdered (m₁ m₂ : Type → Type) (l : Type)
    [Monad m₁] [Monad m₂] [Preorder l] where
  rwp : m₁ α → m₂ β → (α → β → l) → l
  -- + pure / monotonicity / bind laws

/-- eRHL is the `ℝ≥0∞`-valued instance for `OracleComp`.
    pRHL is the same algebra at `l = Prop`. -/
instance [spec₁.Sampleable] [spec₂.Sampleable] :
    MAlgRelOrdered (OracleComp spec₁) (OracleComp spec₂) ℝ≥0∞ where
  rwp := eRelWP
  -- pure / mono / bind discharge from the coupling lemmas
```

[**GPZKS26**] Foundational Multi-Modal Program Verifiers (Loom). POPL 2026

+ extension to relational setting

Denotational Semantics & Program Logic

```
/-- Transformer lifting is automatic: stack `OptionT` on each side, no proof obligations. -/
example : MAlgRelOrdered (OptionT (OracleComp spec₁)) (OptionT (OracleComp spec₂)) ℝ≥0∞ :=
  inferInstance
```

Demonic: returns ⊥ if either execution fails

```
/-- Two-sided state changes the truth lattice from `ℝ≥0∞` to `σ₁ → σ₂ → ℝ≥0∞`. -/
example {σ₁ σ₂ : Type} : MAlgRelOrdered (StateT σ₁ (OracleComp spec₁)) (StateT σ₂ (OracleComp spec₂)) (σ₁ → σ₂ → ℝ≥0∞) :=
  inferInstance
```

[ABDG25] A Quantitative Probabilistic Relational Hoare Logic (eRHL). POPL 2025

```
/-- Relational WP algebra parametric in truth lattice `l`. -/
class MAlgRelOrdered (m₁ m₂ : Type → Type) (l : Type)
    [Monad m₁] [Monad m₂] [Preorder l] where
  rwp : m₁ α → m₂ β → (α → β → l) → l
  -- + pure / monotonicity / bind laws

/-- eRHL is the `ℝ≥0∞`-valued instance for `OracleComp`.
    pRHL is the same algebra at `l = Prop`. -/
instance [spec₁.Sampleable] [spec₂.Sampleable] :
    MAlgRelOrdered (OracleComp spec₁) (OracleComp spec₂) ℝ≥0∞ where
  rwp := eRelWP
  -- pure / mono / bind discharge from the coupling lemmas
```

[GPZKS26] Foundational Multi-Modal Program Verifiers (Loom). POPL 2026

+ extension to relational setting

Tactics for Program Logic

```
/-- `OracleComp` plugs into core `mvcgen` for almost-sure (`Pr[…] = 1`) goals. -/
example (cache₀ : QueryCache spec) : Std.Do.Triple
    (do let _ ← cachingOracle t₁; cachingOracle t₂ : StateT _ (OracleComp spec) _)
    (spred(fun c => ⌜cache₀ ≤ c⌝)) (ψ v c => ⌜cache₀ ≤ c ∧ c t₂ = some v⌝) := by
  mvcgen [cachingOracle_triple] <;> grind
```

```
/-- OTP semantic security via bijection coupling. -/
example (sp : ℕ) (msg₀ msg₁ : BitVec sp) :
    (($ᵗ BitVec sp) >>= fun k => pure (k ^^^ msg₀)) ≡ₚ
    (($ᵗ BitVec sp) >>= fun k => pure (k ^^^ msg₁)) := by
  by_equiv                                                  -- `≡ₚ` ↦ relational triple
  rvcstep using (fun k₁ k₂ => k₂ = k₁ ^^^ msg₀ ^^^ msg₁)    -- couple via key-shift bijection
  -- 3 finishing subgoals (bijection witness + identity coupling on the pure tail)
```

Easycrypt `wp sp seq call inline byequiv byphoare conseq swap …`

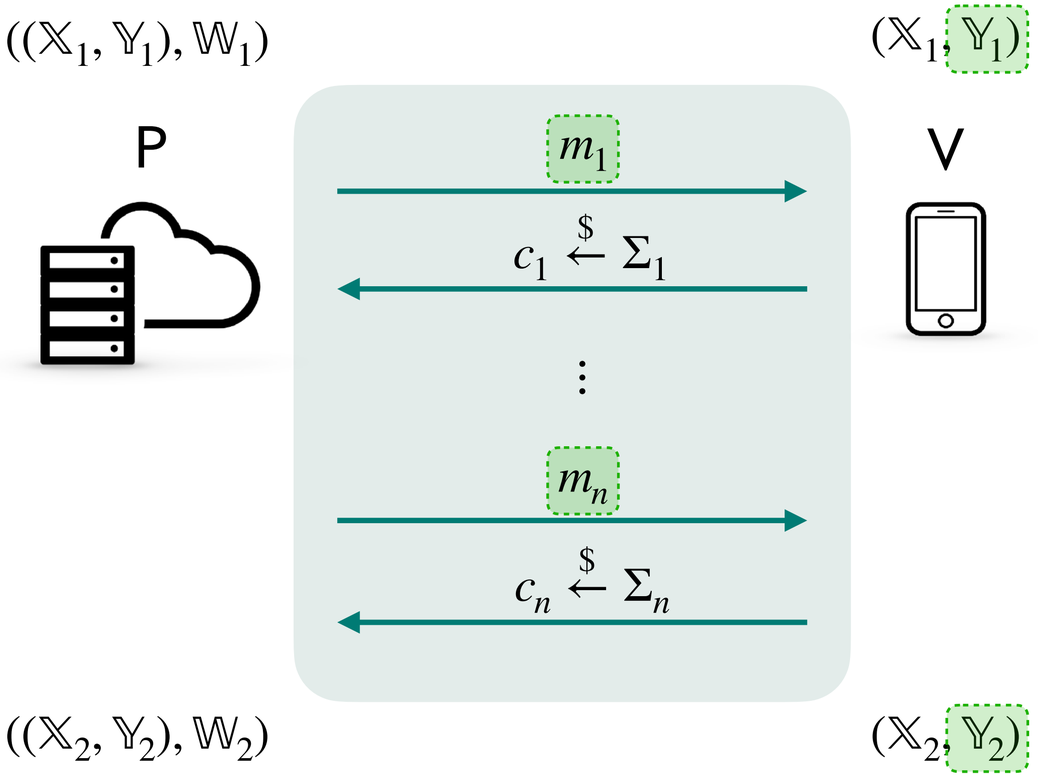
ongoing work, want help!

VCVio `(m/r)vc(gen/step)(?) [using/with/inv] …`

Case Study: Interactive Proof Systems (ArkLib)

Interactive Oracle Reductions:

- Prover & verifier
- Protocol structure may be dependent
- Some messages are opaque to verifier, only interacting through oracle interfaces
- Strong compositional properties
- Complex security games: extractor (knowledge soundness) and simulator (zero-knowledge)



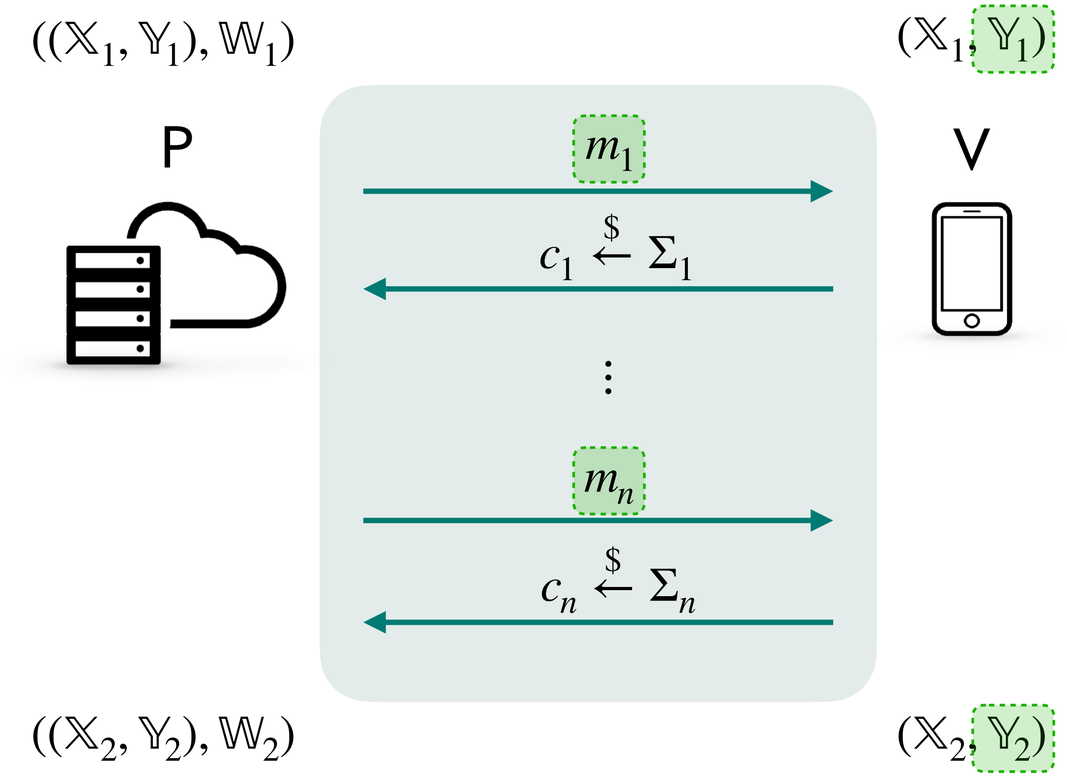
Case Study: Interactive Proof Systems (ArkLib)

```
/-- Canonical protocol spec for oracle reductions. -/
inductive Oracle.Spec : Type 1 where
  | done
  |  «public» (X : Type) (rest : X → Oracle.Spec)
  | oracle   (X : Type) (rest : Oracle.Spec)
```

```
/-- Role decoration: `.public` nodes carry a sender/receiver tag;
    `.oracle` nodes are always sender, so nothing extra is stored. -/
def Oracle.Spec.RoleDeco : Oracle.Spec → Type
  | .done            => PUnit
  | .«public» _ rest => Role × ((x : _) → RoleDeco (rest x))
  | .oracle   _ rest => RoleDeco rest
```

```
/-- Oracle prover type, by recursion on `Oracle.Spec`. -/
def Oracle.Prover (m : Type → Type) : (s : Oracle.Spec) → s.RoleDeco → Type
  | .done,           _              => Unit
  | .«public» X rest, ⟨.sender,   r⟩ => m ((x : X) × Oracle.Prover m (rest x) (r x))
  | .«public» X rest, ⟨.receiver, r⟩ => (x : X) → m (Oracle.Prover m (rest x) (r x))
  | .oracle   X rest, r              => m ((x : X) × Oracle.Prover m rest r)
```

```
/-- Schnorr prover, fully unfolded. (m := OracleComp unifSpec) -/
def prover (sk : F) :
    m ((R : G) × ((c : F) → m (m ((z : F) × PUnit)))) := do
  let r ← $ᵗ F
  pure ⟨r • g, fun c => pure (pure ⟨r + c * sk, ⟨⟩⟩)⟩
```



Case Study: State Separating Proofs

- Proofs about Signal will be many long and complex sequences of games
- While tactics help, we should have all known techniques at our disposal
- State-separating proofs (SSPs) are a recent technique for structuring such games

$\Longrightarrow$ direct motivation for SSProve (which CatCrypt builds upon)

Mom, can we have SSProve?

We have SSProve at home!

Case Study: State Separating Proofs

```
/-- A package exposes export oracles `E`, internally queries imports `I`,
    and keeps private state `σ`. -/
structure Package (I : OracleSpec ιᵢ) (E : OracleSpec ιₑ) (σ : Type) where
  init : σ
  impl : QueryImpl E (StateT σ (OracleComp I))

/-- Run a package against an adversary `A` that queries the exports. -/
def Package.run (P : Package I E σ) (A : OracleComp E α) : OracleComp I α :=
  (simulateQ P.impl A).run' P.init
```

States are separated **by construction** (instead of nominal sets over shared heap)

```
/-- Sequential composition: `outer ∘ inner`. -/
def link (outer : Package M E σ₁) (inner : Package I M σ₂) :
    Package I E (σ₁ × σ₂) := ...

/-- Parallel composition. Each side can only touch its own state factor. -/
def par (p₁ : Package I₁ E₁ σ₁) (p₂ : Package I₂ E₂ σ₂) :
    Package (I₁ + I₂) (E₁ + E₂) (σ₁ × σ₂) := ...
```

Case Study: State Separating Proofs

```
/-- A package exposes export oracles `E`, internally queries imports `I`,
    and keeps private state `σ`. -/
structure Package (I : OracleSpec ιᵢ) (E : OracleSpec ιₑ) (σ : Type) where
  init : σ
  impl : QueryImpl E (StateT σ (OracleComp I))

/-- Run a package against an adversary `A` that queries the exports. -/
def Package.run (P : Package I E σ) (A : OracleComp E α) : OracleComp I α :=
  (simulateQ P.impl A).run' P.init
```

```
/-- The shifted adversary: absorb the outer reduction `P` into the adversary. -/
def shiftLeft (P : Package M E σ₁) (A : OracleComp E α) : OracleComp M α :=
  Prod.fst <$> (simulateQ P.impl A).run P.init

/-- SSP reduction lemma: replacing the inner game in `P.link _` only shifts
    the adversary; the outer reduction `P` becomes part of the adversary. -/
theorem run_link_eq_run_shiftLeft (P : Package M E σ₁) (Q : Package I M σ₂)
    (A : OracleComp E α) :
    (P.link Q).run A = Q.run (P.shiftLeft A)
```

Case Study: ElGamal

(with more being ported)

Case Study: Universal Composability

UC is the **"gold standard"** of cryptography: security under ***arbitrary*** composition

Relevant to Signal Shot! For instance, post-compromise security:

- Global shared state (directory, long-term keys, random oracle)

- Adversarial network scheduling

- Adaptive, momentary corruption with healing

Universally Composable End-to-End Secure Messaging: A Modular Analysis*

Ran Canetti† Palak Jain† Marika Swanberg† Mayank Varia†

May 19, 2023

Existing formalizations (EasyUC & BBGKS21 in Easycrypt, KPC24 in Isabelle) do not come close to stating these properties!

Case Study: Universal Composability

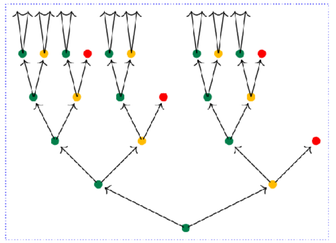
Our approach: build a general theory of open systems, of which UC (and variants) are special cases

```
/-- Interfaces are polynomial functors. -/
structure Interface where
  Port    : Type
  Message : Port → Type

/-- A directed open boundary. -/
structure PortBoundary where
  In  : Interface
  Out : Interface
```

```
/-- The algebra of open systems: a boundary-indexed family
    with three primitive composition operations.
    Closure ≡ `wire` with both outer boundaries empty. -/
structure OpenTheory where
  Obj  : PortBoundary → Type u
  map  : PortBoundary.Hom Δ₁ Δ₂ → Obj Δ₁ → Obj Δ₂
  par  : Obj Δ₁ → Obj Δ₂ → Obj (Δ₁ ⊗ᵇ Δ₂)
  wire : Obj (Δ₁ ⊗ᵇ Γ) → Obj (Γᵛ ⊗ᵇ Δ₂) → Obj (Δ₁ ⊗ᵇ Δ₂)
```

Polynomial Functors: A Mathematical Theory of Interaction



Nelson Niu David I. Spivak

Case Study: Universal Composability

Our approach: build a general theory of open systems, of which UC (and variants) are special cases

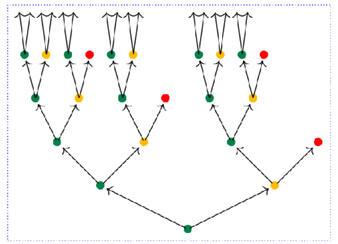
```
variable (T : OpenTheory) (Δ : ProtocolBoundary) (back : PortBoundary)

/-- Closed systems → SPMF distributions; advantage = TV distance. -/
structure Semantics (T : OpenTheory) where
  evalDist : T.Closed → SPMF Unit

abbrev Protocol    := T.Obj (Δ.main ⊗ᵇ Δ.adv)
abbrev Adversary   := T.Obj (Δ.advᵛ  ⊗ᵇ back)
abbrev Environment := T.Obj (Δ.mainᵛ ⊗ᵇ backᵛ)

/-- A minimal UC security definition; operational strength
    (PPT, scheduling, corruption, ...) lives in `T` and `sem`. -/
def UCSecure (sem : Semantics T) (ε : ℝ) (π F : Protocol T Δ) : Prop :=
  ∀ A : Adversary T Δ back,
  ∃ S : Adversary T Δ back,
  ∀ Z : Environment T Δ back,
    sem.distAdvantage
      (T.close (T.wire π A) Z)        -- real-world
      (T.close (T.wire F S) Z) ≤ ε    -- ideal-world
```

Polynomial Functors: A Mathematical Theory of Interaction



Nelson Niu David I. Spivak

Integration with AI Agents

PRs from AI are now commonplace in VCVio & ArkLib:

- Aristotle, Logical Intelligence, unknown

feat(Existence): prove `E_and_Q_BerlekampWelch_condition` ✓
#358 by pitmonticone Contributor was merged on Feb 24 • Approved

feat(QueryPhase): prove `queryKnowledgeStateFunction` ✓
#357 by pitmonticone Contributor was merged on Feb 24 • Approved

feat(Prelude): prove `fixFirstVariablesOfMQP_degreeLE` ✓
#356 by pitmonticone Contributor was merged on Feb 24 • Approved

feat(commitment): layered binding bound via CR chain ✓
#287 by XC0R Contributor was merged 4 days ago

feat(crypto): hash-based commitment + binding ≤ keyed-CR reduction ✓
#286 by XC0R Contributor was merged 4 days ago

feat(hardness): ROM collision resistance + birthday bound ✓
#285 by XC0R Contributor was merged 4 days ago

feat(hardness): collision-resistance hardness assumptions ✓
#283 by XC0R Contributor was merged 4 days ago

Proof for concentration_bounds ✓ automated-proof
#397 by aleph-prover bot was merged on Mar 9 • Approved

Proof for degreeX_mul ✓ automated-proof
#396 by aleph-prover bot was merged on Mar 9 • Approved

Proof for resultant_is_divisible_by_leadingCoeff ✓ automated-proof
#395 by aleph-prover bot was merged on Mar 9 • Approved

Proof for natDeg_sum_eq_of_unique ✓ automated-proof
#394 by aleph-prover bot was merged on Mar 9 • Approved

Integration with AI Agents

PRs from AI are now commonplace in VCVio & ArkLib:

- Aristotle, Logical Intelligence, unknown

Workflows, guidelines, and guardrails are set up for AIs:

- Agents wiki / codebase overview, automated PR review, flagging hygiene & correctness

AI speeds up my development progress significantly:

- Literature search, synthesizing formalization approaches, explaining papers
- Meta-programming (making tactics & widgets), filling in (hard) proofs

Integration with AI Agents

However, not all is dandy…

- Unfocused / autonomous loops may lead nowhere / not most efficient use of tokens
  - Experiment: Forking lemma => Schnorr signature unforgeability
- Getting AIs to use custom tactics is hit-or-miss

How can humans work best with AIs for Signal Shot?

AI speeds up <u>my</u> development progress significantly:

- Literature search, synthesizing formalization approaches, explaining papers
- Meta-programming (making tactics & widgets), filling in (hard) proofs

Summary & Takeaways for Signal Shot

Plan out the foundation, don’t shortcut it!

A case for VCVio (or adaptations thereof into CSLib):

- Expressive oracle handling via connections to algebraic effects & handlers
- Relational program logic, modular derivation via monad transformer stack, with tactics integration
- Base layer for downstream framework (ArkLib) and abstractions (SSP, UC, etc.)

Thank You!